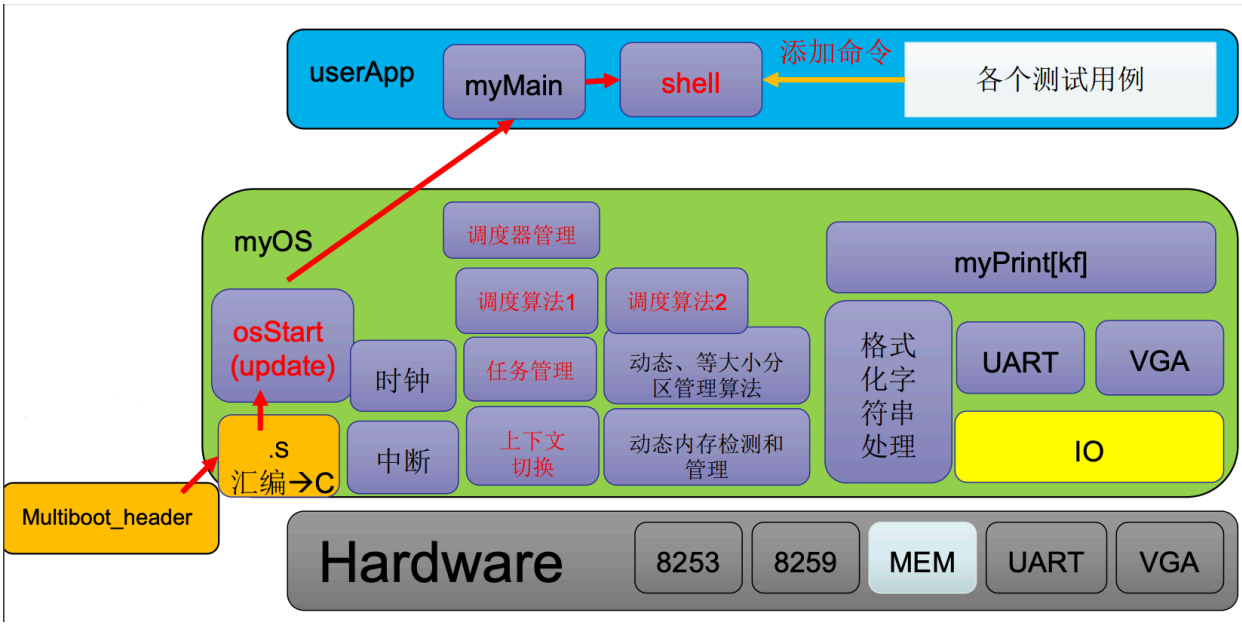
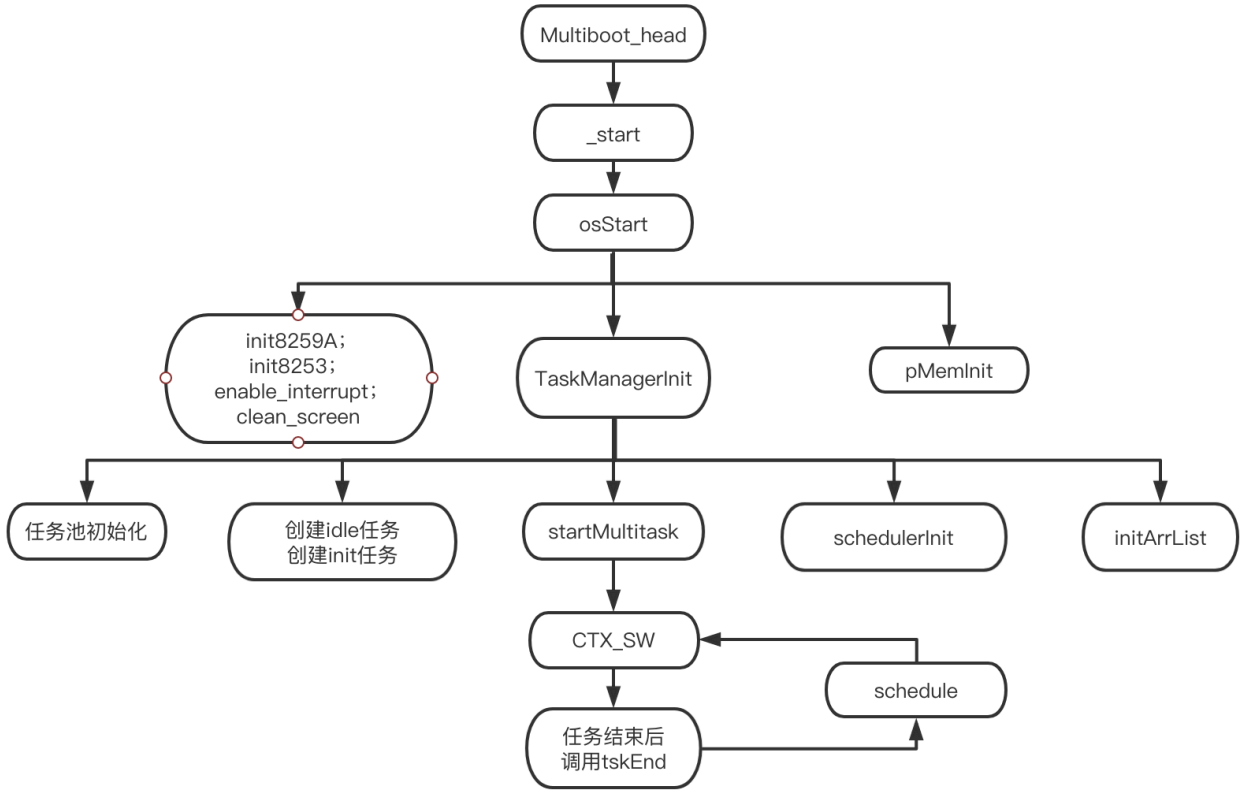
实验六 Task Management

软件框架及概述



概述：从Multiboot_header进入入操作系统内核(myOS)，为进入C程序准备好上下文，初始化操作系统，新增了上下文切换、任务管理、调度器管理和两种调度算法。通过myMain(封装成init任务)进入userApp。userApp中包含若干个测试用例并实现了shell功能(封装成任务)。

主流程及其实现



主流程图

主流程说明：multiboot_header.s调用_start入口，myOS中的Start32.s设置好中断处理，设置时钟中断(包括后续的tick维护、墙钟维护和显示)，提供了osStart()入口，做好第一次调用C语言入口前的准备，进入osStart()后，先初始化操作系统，包括初始化i8259、i8253，并开中断，检测内存并初始化，最后进行任务管理器初始化，包括任务池初始化，手动创建idle任务，初始化到达队列，初始化调度器，创建init任务，最后进入多任务状态从init任务(Mymain)开始执行，任务结束后根据调度器提供的接口调度出下一个任务继续执行，如此往返。

主要功能模块及其实现&源代码说明

实验6的基础来自助教提供的框架

模块一 任务管理

定义任务数据结构如下所示，新增了一个dLink_node型结构体用于维护就绪队列，结构体para指定了任务的优先级、执行时间、到达时间、调度策略等参数。

```c
typedef struct myTCB {
    /* node should be the 1st element*/
    struct dLink_node thisNode;

    /* node body */
    unsigned long state;  // 0:rdy
    int tcbIndex;
    struct myTCB * next;
    unsigned long* stkTop;
    unsigned long stack[STACK_SIZE];
    tskPara para;
    unsigned int leftSlice; // for SCHED_RR or SCHED_RT_RR policy
} myTCB;
```

其他任务管理接口如任务的创建与销毁均同实验5，此处不再展示。

模块二 任务参数管理

```c
void _setTskPara(myTCB *task, tskPara *para){   //将para的中的参数值赋值给task的对应参数

    if(para == (void*)0)
        task->para = defaultTskPara;
    else task->para = *para;
}

void initTskPara(tskPara *buffer){    //将任务buffer的参数设置为默认值
    *buffer = defaultTskPara;
}

void setTskPara(unsigned int option, unsigned int value, tskPara *buffer){   //设置task的设计调度的四个参数
    //option控制buffer的哪个参数要被赋值，value是具体的数值
    switch (option){
        case PRIORITY:        buffer->priority=value; break;
        case EXETIME:         buffer->exeTime=value; break;
        case ARRTIME:         buffer->arrTime=value; break;
        case SCHED_POLICY:    buffer->schedPolicy=value; break;
        default :;
    }
}

void getTskPara(unsigned option, unsigned int *para){      //查看task的设计调度的四个参数
    //option控制buffer的哪个参数要查看 赋值给para
    switch (option){
        case PRIORITY:        *para = currentTsk->para.priority; break;
        case EXETIME:         *para = currentTsk->para.exeTime; break;
        case ARRTIME:         *para = currentTsk->para.arrTime; break;
```

```
        case SCHED_POLICY:    *para = currentTsk->para.schedPolicy; break;
        default :;
    }
}
```

模块三 统一调度接口

根据事先选择好的调度器，封装调度接口。本次实验实现了基于SJF的调度器和基于优先级调度算法的调度器。

```

myTCB *nextTsk(void){                           //返回下一个调度的任务
    return sysScheduler->nextTsk_func();
}

void enqueueTsk(myTCB *tsk){                    //任务进入就绪队列
    sysScheduler->enqueueTsk_func(tsk);
}

void dequeueTsk(myTCB *tsk){                    //任务出就绪队列
    sysScheduler->dequeueTsk_func(tsk);
}

void createTsk_hook(myTCB *created){            //创建任务时的hook
    if(sysScheduler->createTsk_hook)
        sysScheduler->createTsk_hook(created);
}

extern void scheduler_hook_main(void);

void schedulerInit(void){                       //调度器初始化
    scheduler_hook_main();
    sysScheduler->schedulerInit_func();
}

void scheduler_tick(void){                      //tick时的hook
    if(sysScheduler->tick_hook)
        sysScheduler->tick_hook();
}

void schedule(void){                            //实现调度功能
    static int idle_times=0;
    myTCB * prevTsk;

    disable_interrupt();

    prevTsk = currentTsk;
    currentTsk = sysScheduler->nextTsk_func();
    if(currentTsk == idleTsk && idle_times==0)     //第一次执行idle
```

```
    {
        idle_times++;
        context_switch(prevTsk,currentTsk);
    }
    else if(currentTsk == idleTsk && prevTsk == idleTsk);
    else  context_switch(prevTsk,currentTsk);

    enable_interrupt();
}
```

模块四 调度器管理

```
unsigned int getSysScheduler(void){     //返回调度器类型
    return sysScheduler->type;
}

void setSysScheduler(unsigned int what){    //设置*sysScheduler选择使用哪种调度器
    switch (what){
        case SCHEDULER_FCFS:        sysScheduler = &scheduler_FCFS; break;
        case SCHEDULER_SJF:         sysScheduler = &scheduler_SJF; break;
        case SCHEDULER_PRIORITY0:   sysScheduler = &scheduler_PRIO0; break;
        //case SCHEDULER_RR:          sysScheduler = &scheduler_RR; break;
        //case SCHEDULER_PRIORITY:    sysScheduler = &scheduler_PRIO; break;
        //case SCHEDULER_MQ:          sysScheduler = &scheduler_MQ; break;
        //case SCHEDULER_FMQ:         sysScheduler = &scheduler_FMQ; break;
        default :;
    }
}

void getSysSchedulerPara(unsigned int who, unsigned int *para){     //返回调度器参数
    switch (who){
        case SCHED_RR_SLICE: *para = defaultSlice; break;
        case SCHED_RT_RR_SLICE: *para = defaultRtSlice; break;
        default :;
    }

}

void setSysSchedulerPara(unsigned int who, unsigned int para){      //设置调度器参数
    switch (who){
        case SCHED_RR_SLICE: defaultSlice = para; break;
        case SCHED_RT_RR_SLICE: defaultRtSlice = para; break;
        default :;
```

```
    }
}
```

模块五 Priority0调度算法

相比FCFS算法，区别主要体现在任务入队列接口**tskEnqueuePRIO0**中，任务并不是直接挂在队尾，而是根据优先级大小找到应该插在队列中的位置。

```
myTCB *rqPRIO0;  //优先级调度队列

void rqPRIO0Init(void) {         //初始化优先级调度队列
    rqPRIO0 = idleTsk;  // head <- idleTsk
    dLinkListInit((dLinkedList *)(&(rqPRIO0->thisNode)));
}

myTCB * nextPRIO0Tsk(void) {     //返回队列中下一个待调度的任务
    return (myTCB*)dLinkListFirstNode((dLinkedList*)rqPRIO0);
}


void tskEnqueuePRIO0(myTCB *tsk){            //任务入队列
    dLink_node *q = ((dLinkedList*)rqPRIO0)->next;
    while(q!=(dLinkedList*)rqPRIO0 && ((myTCB *)q)->para.priority<tsk->para.priority)   //按优先级从小到大顺序入队
        q=q->next;
    dLinkInsertBefore((dLinkedList*)rqPRIO0,q,(dLink_node *)tsk);
}


void tskDequeuePRIO0(myTCB *tsk){            //任务出队列
    dLinkDelete((dLinkedList*)rqPRIO0,(dLink_node*)tsk);
}


void schedulerInit_PRIO0(void){             //调度器初始化
    rqPRIO0Init();

    /* default for all task except idleTsk*/
    defaultTskPara.schedPolicy = SCHED_PRIO;

    /* special for idleTsk*/
    _setTskPara(idleTsk,&defaultTskPara);
    idleTsk->para.schedPolicy = SCHED_IDLE;

    initTsk_para.priority = 0;             //init任务优先级设为0，为最高优先级
}

struct scheduler scheduler_PRIO0 = {       //将各接口封装成一个调度器
    .type = SCHEDULER_PRIORITY0,
```

```
    .nextTsk_func = nextPRIO0Tsk,
    .enqueueTsk_func = tskEnqueuePRIO0,
    .dequeueTsk_func = tskDequeuePRIO0,
    .schedulerInit_func = schedulerInit_PRIO0,
    .createTsk_hook = NULL,
    .tick_hook = NULL
};
```

模块六 SJF(非抢占式)调度算法

相当于以执行时间作为优先级的调度算法。主要新增了一个**tickSJF_hook**机制在每次tick时判断当前任务是否已超过预计的执行时间，若超过则强行结束该任务重新调度。

```
myTCB *rqSJF;  //SJF队列

void rqSJFInit(void) {           //初始化SJF调度队列
    rqSJF = idleTsk;    // head <- idleTsk
    dLinkListInit((dLinkedList *)(&(rqSJF->thisNode)));
}

int Start_tick;              //记录任务开始的时刻
extern int getTick(void);
extern void tskEnd(void);

myTCB * nextSJFTsk(void) {       //返回队列中下一个待调度的任务
    Start_tick = getTick();
    return (myTCB*)dLinkListFirstNode((dLinkedList*)rqSJF);
}


void tskEnqueueSJF(myTCB *tsk){          //任务入队列
    dLink_node *q = ((dLinkedList*)rqSJF)->next;
    while(q!=(dLinkedList*)rqSJF && ((myTCB *)q)->para.exeTime<=tsk->para.exeTime)
         q=q->next;
    dLinkInsertBefore((dLinkedList*)rqSJF,q,(dLink_node *)tsk);
}


void tskDequeueSJF(myTCB *tsk){          //任务出队列
    dLinkDelete((dLinkedList*)rqSJF,(dLink_node*)tsk);
}


void schedulerInit_SJF(void){            //调度器初始化
    rqSJFInit();

    /* default for all task except idleTsk*/
    defaultTskPara.schedPolicy = SCHED_SJF;
```

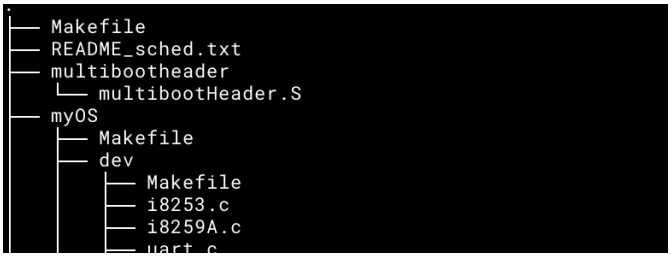
```
36
37      /* special for idleTsk*/
38      _setTskPara(idleTsk,&defaultTskPara);
39      idleTsk->para.schedPolicy = SCHED_IDLE;
40
41      initTsk_para.exeTime = MAX_EXETIME;     //init任务执行时间设为MAX以保证不被强行中断
42  }
43
44  void tickSJF_hook(void){                    //每次tick检查一次是否超时
45      if(getTick()-Start_tick>=currentTsk->para.exeTime)
46          tskEnd();
47  }
48
49  struct scheduler scheduler_SJF = {          //将各接口封装成一个调度器
50      .type = SCHEDULER_SJF,
51      .nextTsk_func = nextSJFTsk,
52      .enqueueTsk_func = tskEnqueueSJF,
53      .dequeueTsk_func = tskDequeueSJF,
54      .schedulerInit_func = schedulerInit_SJF,
55      .createTsk_hook = NULL,
56      .tick_hook = tickSJF_hook
57  };
```

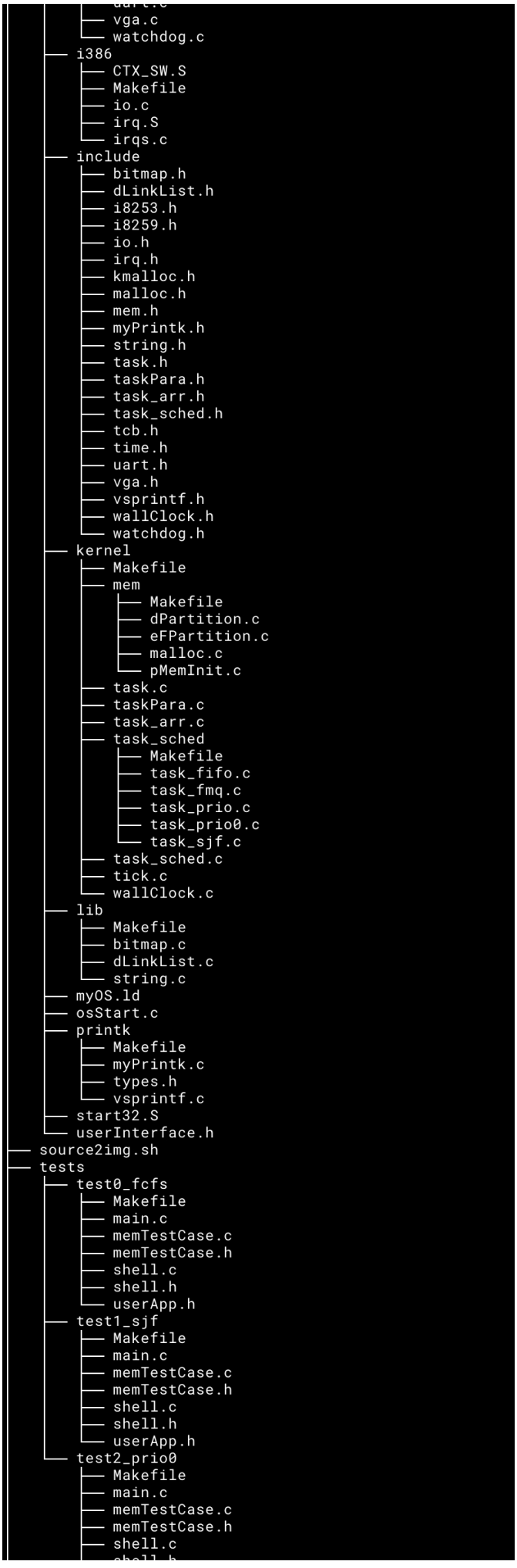
一些其他模块

到达队列的入队接口，根据ARRTIME的大小排序决定任务在队列中的位置。

```
1   /* arrTime: small --> big */
2   void ArrListEnqueue(myTCB* tsk){      //根据tsk新建一个节点 按照arrTime小到大的顺序插入到链表的对应位置
3       arrNode *p = tcb2Arr(tsk);
4       p->theTCB = tsk;
5       p->arrTime = tsk->para.arrTime;
6       dLink_node *q = arrList.next;
7       while(q!=&arrList && ((arrNode *)q)->arrTime<=p->arrTime)
8           q=q->next;
9       dLinkInsertBefore(&arrList,q,(dLink_node *)p);
10  }
```

目录组织




├── uart.c
├── vga.c
└── watchdog.c
├── i386
├── CTX_SW.S
├── Makefile
├── io.c
├── irq.S
└── irqs.c
├── include
├── bitmap.h
├── dLinkList.h
├── i8253.h
├── i8259.h
├── io.h
├── irq.h
├── kmalloc.h
├── malloc.h
├── mem.h
├── myPrintk.h
├── string.h
├── task.h
├── taskPara.h
├── task_arr.h
├── task_sched.h
├── tcb.h
├── time.h
├── uart.h
├── vga.h
├── vsprintf.h
├── wallClock.h
└── watchdog.h
├── kernel
├── Makefile
├── mem
├── Makefile
├── dPartition.c
├── eFPartition.c
├── malloc.c
└── pMemInit.c
├── task.c
├── taskPara.c
├── task_arr.c
├── task_sched
├── Makefile
├── task_fifo.c
├── task_fmq.c
├── task_prio.c
├── task_prio0.c
└── task_sjf.c
├── task_sched.c
├── tick.c
└── wallClock.c
├── lib
├── Makefile
├── bitmap.c
├── dLinkList.c
└── string.c
├── myOS.ld
├── osStart.c
├── printk
├── Makefile
├── myPrintk.c
├── types.h
└── vsprintf.c
├── start32.S
└── userInterface.h
├── source2img.sh
├── tests
├── test0_fcfs
├── Makefile
├── main.c
├── memTestCase.c
├── memTestCase.h
├── shell.c
├── shell.h
└── userApp.h
├── test1_sjf
├── Makefile
├── main.c
├── memTestCase.c
├── memTestCase.h
├── shell.c
├── shell.h
└── userApp.h
└── test2_prio0
├── Makefile
├── main.c
├── memTestCase.c
├── memTestCase.h
├── shell.c

Makefile组织

关键规则：

```
output/myOS.elf: ${OS_OBJS} ${MULTI_BOOT_HEADER}
    ${CROSS_COMPILE}ld -n -T myOS/myOS.ld ${MULTI_BOOT_HEADER} ${OS_OBJS}
-o output/myOS.elf

output/%.o : %.S      #所有的.s生成.o
    @mkdir -p $(dir $@)
    @${CROSS_COMPILE}gcc ${ASM_FLAGS} -c -o $@ $<

output/%.o : %.c      #所有的.c生成.o
    @mkdir -p $(dir $@)
    @${CROSS_COMPILE}gcc ${C_FLAGS} -c -o $@ $<
```

先由各级子目录下的.c 和.s文件生成.o文件，再将.o文件作为依赖文件，按照myOS.ld规则链接成终极目标文件myOS.elf。

代码布局说明

首先定位到内存中1M地址处。可执行文件的.text段从此处开始。先存放.multiboot_header段[12字节]，往后对齐8字节后，再存放所有输入文件的.text段。 往后对齐16字节，开始存放可执行文件的.data段，即为所有输入文件的.data段。往后对齐16字节，接着存放可执行文件的.bss段，包括所有输入文件中未初始化的全局变量。bss段结束后再向后对齐16字节，此处以_end作为结束标记。往后对齐512字节。

编译过程说明

分别直接在终端运行./source2img.sh test1_sjf 和 ./source2img.sh test2_prio0对两种不同调度算法进行编译，并将串口重定向到伪终端，运行时会告知具体是哪个，并据此输入

```
sudo screen /dev/pts/0   #假设是/dev/pts/0
```

接着就可以通过伪终端输入命令。

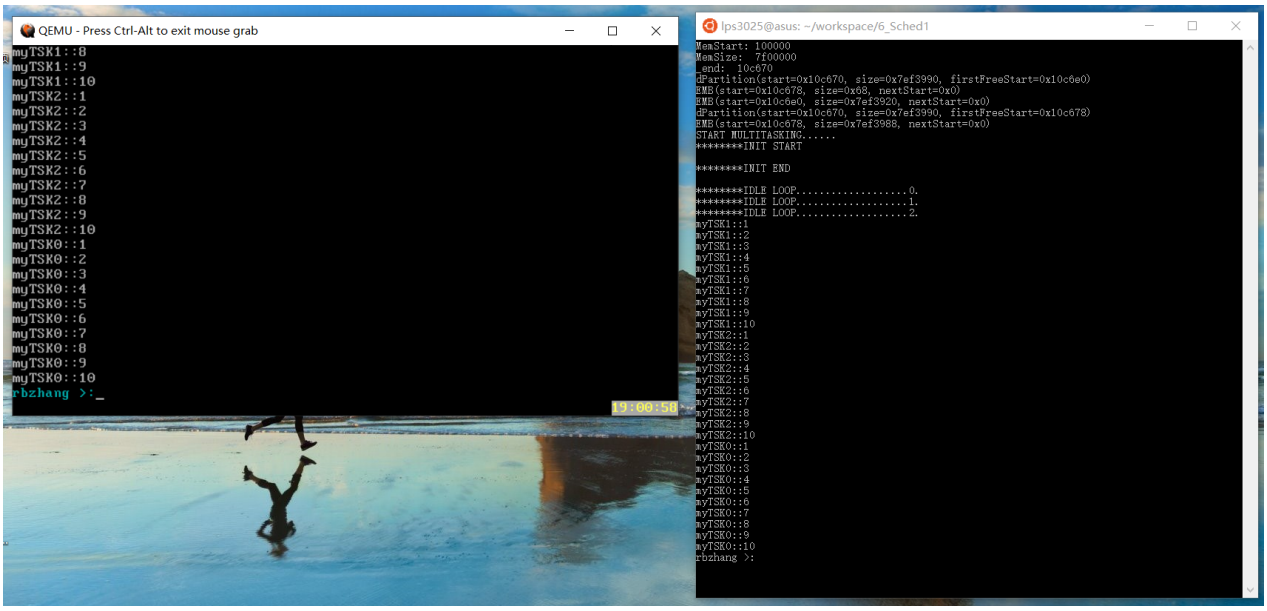
运行和运行结果说明

Priority0调度算法的测试用例主要设置如下：

```
void scheduler_hook_main(void){
    //prior settings
    setSysScheduler(SCHEDULER_PRIORITY0);
}

void doSomeTestBefore(void){
    setWallClock(18,59,59);     //set time 18:59:59
    setWallClockHook(&wallClock_hook_main);
}

void myTSK0(void){
    int j=1;
    while(j<=10){
        myPrintf(0x7,"myTSK0::%d    \n",j);
        busy_n_ms(120);
        j++;
    }
    tskEnd();   //the task is end
}

void myTSK1(void){
    int j=1;
    while(j<=10){
        myPrintf(0x7,"myTSK1::%d    \n",j);
        busy_n_ms(120);
        j++;
    }
    tskEnd();   //the task is end
}

void myTSK2(void){
    int j=1;
    while(j<=10){
        myPrintf(0x7,"myTSK2::%d    \n",j);
        busy_n_ms(120);
        j++;
    }
    tskEnd();   //the task is end
}

void testSchedulerPRIORITY0(void){  //FCFS or RR or SJF or PRIORITY0
    tskPara tskParas[4];
    int i;

    if(getSysScheduler()!=SCHEDULER_PRIORITY0) {
        myPrintf(0x3,"NEED scheduler:
SCHEDULER_PRIORITY0!!!!!!STOPED!!!!!!!!");
        return;
    }
```

```
49
50      for(i=0;i<4;i++) initTskPara(&tskParas[i]);
51
52      setTskPara(ARRTIME,250,&tskParas[0]);
53      setTskPara(PRIORITY,10,&tskParas[0]);
54      createTsk(myTSK0,&tskParas[0]);
55
56      setTskPara(ARRTIME,250,&tskParas[1]);
57      setTskPara(PRIORITY,5,&tskParas[1]);
58      createTsk(myTSK1,&tskParas[1]);
59
60      setTskPara(ARRTIME,250,&tskParas[2]);
61      setTskPara(PRIORITY,7,&tskParas[2]);
62      createTsk(myTSK2,&tskParas[2]);
63
64      initShell();
65      memTestCaseInit();
66      setTskPara(ARRTIME,520,&tskParas[3]);
67      setTskPara(PRIORITY,12,&tskParas[3]);
68      createTsk(startShell,&tskParas[3]);  //    startShell();
69  }
70
71  void myMain(void){    //main is our init task
72      clear_screen();
73
74      doSomeTestBefore();
75
76      myPrintf(0x7,"********INIT START\n\n");
77      testSchedulerPRIORITY0();
78      myPrintf(0x7,"********INIT END\n\n");
79
80      tskEnd();  // init end
81  }
```

运行结果如下：



运行结果说明：myTSK0、myTSK1、myTSK2的优先级分别设为10、5、7(优先级越小越优先)，到达时间均设为250。由上图可见在经过一开始的idle后，按照优先级从小到大顺序执行了myTSK1、myTSK2、myTSK0。最后执行达到最晚且优先级最低的shell。

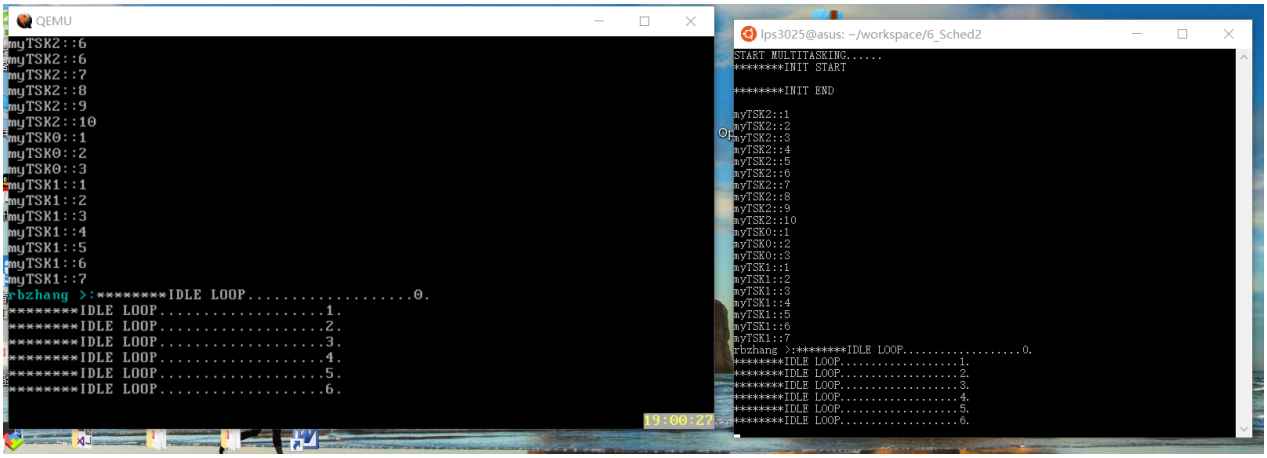
SJF调度算法的测试用例主要设置如下：

```
void scheduler_hook_main(void){
    //prior settings
    setSysScheduler(SCHEDULER_SJF);
}

void doSomeTestBefore(void){
    setWallClock(18,59,59);     //set time 18:59:59
    setWallClockHook(&wallClock_hook_main);
}

void myTSK0(void){
    int j=1;
    while(j<=10){
        myPrintf(0x7,"myTSK0::%d    \n",j);
        busy_n_ms(120);
        j++;
    }
    tskEnd();   //the task is end
}

void myTSK1(void){
    int j=1;
    while(j<=10){
        myPrintf(0x7,"myTSK1::%d    \n",j);
        busy_n_ms(120);
        j++;
```

```
    }
    tskEnd();   //the task is end
}

void myTSK2(void){
    int j=1;
    while(j<=10){
        myPrintf(0x7,"myTSK2::%d    \n",j);
        busy_n_ms(120);
        j++;
    }
    tskEnd();   //the task is end
}

void testSchedulerSJF(void){  //FCFS or RR or SJF or PRIORITY0
    tskPara tskParas[4];
    int i;

    if(getSysScheduler()!=SCHEDULER_SJF) {
        myPrintf(0x3,"NEED scheduler: SCHEDULER_SJF!!!!!!STOPED!!!!!!!!");
        return;
    }

    for(i=0;i<4;i++) initTskPara(&tskParas[i]);

    setTskPara(ARRTIME,50,&tskParas[0]);
    setTskPara(EXETIME,20,&tskParas[0]);
    createTsk(myTSK0,&tskParas[0]);

    setTskPara(ARRTIME,100,&tskParas[1]);
    setTskPara(EXETIME,50,&tskParas[1]);
    createTsk(myTSK1,&tskParas[1]);

    setTskPara(ARRTIME,0,&tskParas[2]);
    setTskPara(EXETIME,80,&tskParas[2]);
    createTsk(myTSK2,&tskParas[2]);

    initShell();
    memTestCaseInit();
    setTskPara(ARRTIME,120,&tskParas[3]);
    setTskPara(EXETIME,2000,&tskParas[3]);
    createTsk(startShell,&tskParas[3]);  //    startShell();
}

void myMain(void){    //main is our init task
    clear_screen();

    doSomeTestBefore();

```

```
76      myPrintf(0x7,"********INIT START\n\n");
77      testSchedulerSJF();
78      myPrintf(0x7,"********INIT END\n\n");
79
80      tskEnd();  // init end
81  }
```

运行结果如下：



运行结果说明：myTSK0、myTSK1、myTSK2的执行时间分别设为20、50、80，到达时间设为50、100、0。由上图可见，最初先执行唯一到达的myTSK2，且myTSK2设置了足够的执行时间可完成该任务。myTSK2完成后myTSK0、myTSK1均已就绪，而执行时间更小的myTSK0先被调度，执行了3次循环后就已经过了预计的执行时间 因而被强行终止，接着执行myTSK1。myTSK1也因为执行时间不足被强行终止，最后执行shell。在shell用完了2000的执行时间后，调度出idle任务继续一直执行下去。

遇到的问题和解决方案说明

一开始纠结了下idle和schedule的相互调用关系。后来明确了idle_times这个变量的用处后就写出来了。

为了实现SJF需要知道每个任务开始执行的时间。为此设了一个全局变量StartTime在每个任务刚被调度时记录当时的ticknumber。